Análisis y Diseño de Algoritmos

Programación Dinámica

DR. JESÚS A. GONZÁLEZ BERNAL
CIENCIAS COMPUTACIONALES
INAOE

Introducción a Programación Dinámica

- Parecido a divide y conquista
  - Resuelve problemas combinando soluciones
  - Programación se refiere a resolver problemas en forma tabular
  - Programación dinámica aplica cuando los subproblemas no son independientes
    - Comparten subproblemas
  - Resuelve cada subproblema sólo una vez, guarda la solución, ahorra tiempo
- Generalmente utilizada en problemas de optimización

Introducción a Programación Dinámica

- Cuatro pasos
  1. Caracterizar la estructura de una solución óptima
  2. Recursivamente definir el valor de una solución óptima
  3. Calcular el valor de una solución óptima de un modo bottom-up
  4. Construir una solución óptima a partir de la información calculada

Multiplicación de Matrices

- Dadas 2 matrices:
  - $A_{pxq} * B_{qxr} = C_{pxr}$, $A_{2x3} * B_{3x2} = C_{2x2}$
  - Se requieren p x q x r multiplicaciones

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} * \begin{bmatrix} 7 & 8 \\ 9 & 10 \\ 11 & 12 \end{bmatrix} = \begin{bmatrix} 1x7+2x9+3x11 & 1x8+2x10+3x12 \\ 4x7+5x9+6x11 & 4x8+5x10+6x12 \end{bmatrix}$$

```
MATRIX-MULTIPLY(A, B)
1  if columns[A] ≠ rows[B]
2     then error "incompatible dimensions"
3     else  for i ← 1 to rows[A]
4                 do for j ← 1 to columns[B]
5                       do C[i, j] ← 0
6                          for k ← 1 to columns[A]
7                              do C[i, j] ← C[i, j] + A[i, k] · B[k, j]
8           return C
```

Multiplicación de Matrices en Cadena

- Entrada: una cadena de n matrices <*A*1, *A*2, …, *An*>
- Salida: el producto de las matrices *A*1*A*2…*An*.
- Algoritmo
  - Acomodar los paréntesis a manera de minimizar el número de productos escalares al multiplicar las matrices
  - $A_1A_2A_3$ se puede agrupar como:
    - $(A_1A_2)A_3$, ó como $A_1(A_2A_3)$

Ejemplo

- Si A_1 es de 10 x 100, A_2 de 100 x 5 y A_3 de 5 x 50 →
  - $A_1 (A_2 A_3)$ → 100x5x50 + 10x100x50 = 25,000 + 50,000 = 75,000 multiplicaciones, ($A_2 A_3$ es una matriz de 100x50)
  - $(A_1 A_2) A_3$ → 10x100x5 + 10x5x 50 = 5,000 + 2,500 = 7,500 multiplicaciones ($A_1 A_2$ es una matriz de 10x5)

Solución por Fuerza Bruta

- Intentamos resolver el problema probando todas las maneras de agrupar con paréntesis
- No es una solución eficiente
- Sea $P(n)$ el número de formas diferentes de acomodar los paréntesis en una secuencia de n matrices
- Tenemos la recurrencia:

$$P(n) = \begin{cases} 1 & \text{if } n = 1, \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{if } n \geq 2. \end{cases}$$

- (la secuencia de los números de catalán)
- $P(n) = C(n\text{-}1)$, donde:

$$C(n) = \frac{1}{n+1}\binom{2n}{n} = \Omega\left(\frac{4^n}{n^{3/2}}\right)$$

  - El número de soluciones es exponencial en n
    - → Resolver por programación dinámica

Paso 1: Caracterizar la Estructura de la Solución Óptima (La estructura para agrupar los paréntesis)

- Sea $A_{i..j}$ la matriz resultante del producto $A_iA_{i+1}...A_j$ donde $i<j$
- Si se divide el producto entre A_k y A_{k+1} para $i \leq k < j$
- Se calcula por separado $A_{i..k}$ y $A_{k+1..j}$
  - La solución a cada subproblema debe ser óptima para que la solución de $A_1..A_n$ sea óptima
  - Costo = Costo($A_{i..k}$) + Costo($A_{k+1..j}$) + Costo de multiplicar ambas matrices
- Si hubiera otra forma de agrupar que nos de mejor costo entonces la anterior no sería la óptima

Paso 1: Caracterizar la Estructura de la Solución Óptima (La estructura para agrupar los paréntesis)

- Subestructura óptima:
  - Construir soluciones óptimas para todos los subproblemas (así trabaja programación dinámica)
    - Por eso se llama subestructura óptima

Paso 2. Definir una Solución Recursiva

- Definimos costo de una solución óptima recursivamente en términos de la solución óptima a subproblemas
- Subproblemas
  - Problema de determinar el costo mínimo de agrupar las matrices con paréntesis para $A_i A_{i+1} \ldots A_j$ para $1 \leq i \leq j \leq n$
  - Sea $m[i,j]$ el número mínimo de multiplicaciones escaleres para calcular $A_{i..j}$
    - El costo total para obtener $A_{1..n}$ sería $m[1,n]$

Paso 2. Definir una Solución Recursiva

- Definimos $m[i,j]$
  - Si $i = j$, $m[i,j] = 0$ (problema trivial, una sóla matriz, no son necesarias multiplicaciones de escalares)
  - Si $i < j$, asumimos una división óptima entre A_k y A_{k+1} ($i \leq k < j$)

    $m[i,j]$ = costo de calcular $A_{i..k}$ + costo de calcular $A_{k+1..j}$ + costo de calcular $A_{i..k}A_{k+1..j}$

    $= m[i,k] + m[k+1,j] + p_{i-1}p_k p_j$

    Sin embargo, no conocemos el valor de k e intentaremos todas las j-i posibilidades
  - La definición recursiva para el mínimo costo de agrupar los paréntesis del producto $A_i A_{i+1} \ldots A_j$ es:

$$m[i, j] = \begin{cases} 0 & \text{if } i = j, \\ \min\limits_{i \leq k < j} \{m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j\} & \text{if } i < j. \end{cases}$$

Paso 3: Calculando los Costos Óptimos

- Podemos utilizar la recurrencia anterior para calcular el costo mínimo de $m[1,n]$ para multiplicar $A_1A_2...A_n$
  - Pero el algoritmo todavía es exponencial (no mejor que fuerza bruta)
  - Algo bueno es que tenemos relativamente pocos subproblemas
    - Uno para cada elección de i y j donde $1 \le i \le j \le n$, ó

$$\binom{n}{2} + n = \Theta(n^2)$$

    - El algoritmo puede encontrar subproblemas repetidos
      - Traslape (Overlapping)
      - Utilizamos método bottom-up tabular → paso 3

Paso 3: Calculando los Costos Óptimos

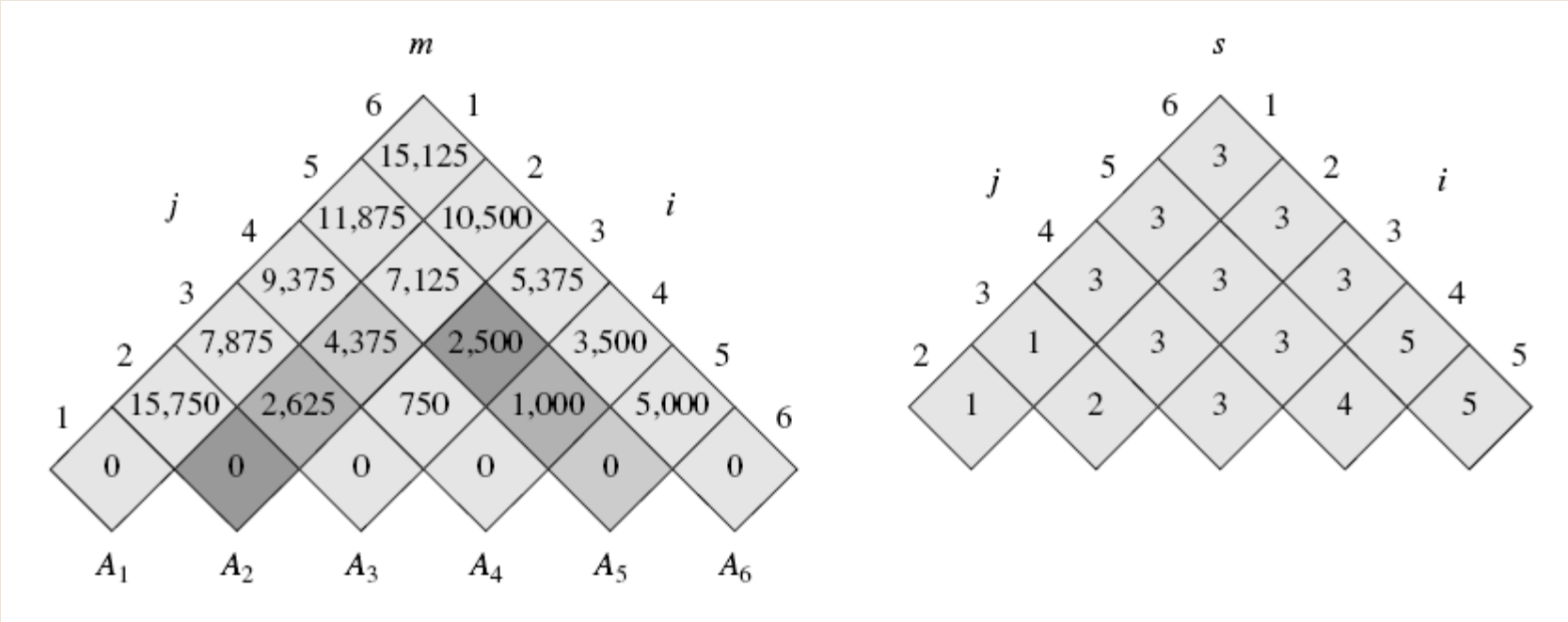
- Método bottom-up
  - Resolvemos subproblemas pequeños primero y los más grandes serán más fáciles de resolver
- Definimos 2 arreglos
  - $m[1..n, 1..n]$, para almacenar costos mínimos
  - $s[1..n, 1..n]$, para almacenar las divisiones óptimas, índice de k
    - Para construir la solución óptima

Paso 3: Calculando los Costos Óptimos

- Tiempo de ejecución de $O(n^3)$ y requiere $\Theta(n^2)$ memoria para almacenar m y s.

```
MATRIX-CHAIN-ORDER(p)
 1  n ← length[p] − 1
 2  for i ← 1 to n
 3       do m[i, i] ← 0
 4  for l ← 2 to n          ▷ l is the chain length.
 5       do for i ← 1 to n − l + 1
 6            do j ← i + l − 1
 7               m[i, j] ← ∞
 8               for k ← i to j − 1
 9                    do q ← m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
10                       if q < m[i, j]
11                          then m[i, j] ← q
12                               s[i, j] ← k
13  return m and s
```

Paso 3: Calculando los Costos Óptimos



Paso 3: Calculando los Costos Óptimos

Figure 15.3 The m and s tables computed by MATRIX-CHAIN-ORDER for $n = 6$ and the following matrix dimensions:

matrix	dimension
A_1	30×35
A_2	35×15
A_3	15×5
A_4	5×10
A_5	10×20
A_6	20×25

The tables are rotated so that the main diagonal runs horizontally. Only the main diagonal and upper triangle are used in the m table, and only the upper triangle is used in the s table. The minimum number of scalar multiplications to multiply the 6 matrices is $m[1, 6] = 15{,}125$. Of the darker entries, the pairs that have the same shading are taken together in line 9 when computing

$$
\begin{aligned}
m[2, 5] &= \min \begin{cases} m[2, 2] + m[3, 5] + p_1 p_2 p_5 = 0 + 2500 + 35 \cdot 15 \cdot 20 &= 13000\,, \\ m[2, 3] + m[4, 5] + p_1 p_3 p_5 = 2625 + 1000 + 35 \cdot 5 \cdot 20 &= 7125\,, \\ m[2, 4] + m[5, 5] + p_1 p_4 p_5 = 4375 + 0 + 35 \cdot 10 \cdot 20 &= 11375 \end{cases} \\
&= 7125\,.
\end{aligned}
$$

Paso 4. Construyendo la Solución Óptima

- MATRIX-CHAIN-ORDER encuentra el número óptimo de multiplicaciones escalares
- Construimos la solución óptima con la información en $s[1..n, 1..n]$

```
PRINT-OPTIMAL-PARENS(s, i, j)
1  if i = j
2     then print "A"_i
3     else print "("
4          PRINT-OPTIMAL-PARENS(s, i, s[i, j])
5          PRINT-OPTIMAL-PARENS(s, s[i, j] + 1, j)
6          print ")"
```

In the example of Figure 15.3, the call PRINT-OPTIMAL-PARENS$(s, 1, 6)$ prints the parenthesization $((A_1(A_2A_3))((A_4A_5)A_6))$.

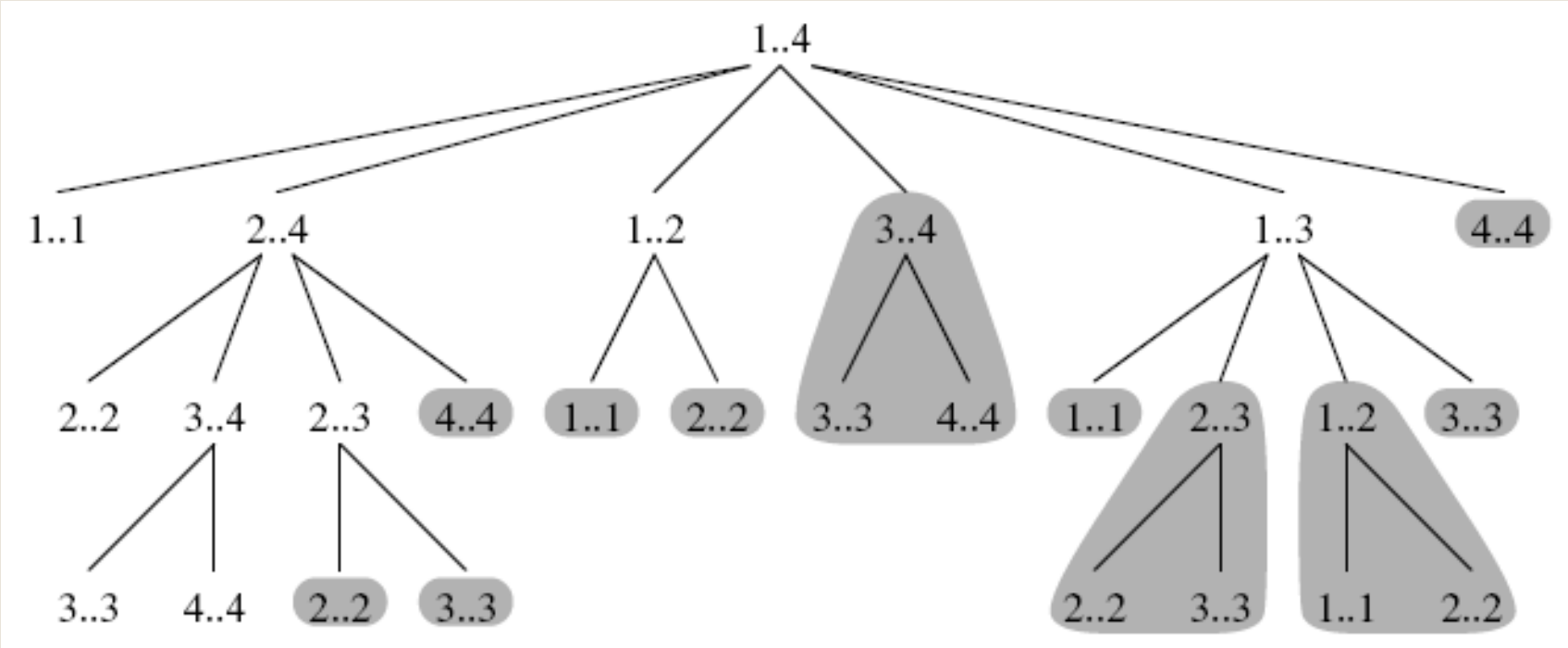
Elementos de la Programación Dinámica

- Subestructura óptima
  - Un problema con solución óptima que tiene sub-problemas con soluciones óptimas
  - Si se presenta esta propiedad, podría aplicar (probablemente) programación dinámica

Elementos de la Programación Dinámica

- Problemas traslapados
  - El espacio de sub-problemas debe ser pequeño
  - Un alg. recursivo los resolvería muchas veces
    - Lo ideal (recursivo) sería sólo generar/resolver problemas nuevos
  - Generalmente el número de sub-problemas diferentes es polinomial en tamaño de la entrada
- Divide y conquista genera nuevos problemas cada paso de la recursión

Traslape en Multiplicación en Cadena de Matrices



Algoritmo Recursivo para Multiplicación de Matrices en Cadena

$$m[i, j] = \begin{cases} 0 & \text{if } i = j , \\ \min\limits_{i \le k < j} \{m[i, k] + m[k+1, j] + p_{i-1} p_k p_j\} & \text{if } i < j . \end{cases}$$

```
RECURSIVE-MATRIX-CHAIN(p, i, j)
1  if i = j
2     then return 0
3  m[i, j] ← ∞
4  for k ← i to j − 1
5        do q ← RECURSIVE-MATRIX-CHAIN(p, i, k)
                + RECURSIVE-MATRIX-CHAIN(p, k + 1, j)
                + p_{i−1} p_k p_j
6           if q < m[i, j]
7              then m[i, j] ← q
8  return m[i, j]
```

Análisis de la Solución Recursiva

$$T(n) = \begin{cases} \Theta(1) & n = 1 \\ \Theta(1) + \sum_{k=1}^{n-1} (T(k) + T(n-k) + \Theta(1)) & n > 1 \end{cases}$$

$$T(n) = \Theta(1) + \sum_{k=1}^{n-1} (T(k) + T(n-k) + \Theta(1))$$

$$= \Theta(1) + \sum_{k=1}^{n-1} \Theta(1) + \sum_{k=1}^{n-1} T(k) \sum_{k=1}^{n-1} T(n-k)$$

$$= \Theta(1) + \Theta(n-1) + \sum_{k=1}^{n-1} T(k) + \sum_{k=1}^{n-1} T(k)$$

$$= \Theta(n) + 2\sum_{k=1}^{n-1} T(k)$$

Análisis de la Solución Recursiva

- Por el método de substitución, probando que $T(n)=\Omega(2^n)$
  - Mostrar: $T(n) = \Omega(2^n) \geq c2^n$
  - Asumiendo: $T(k) \geq c2^k$ para $k < n$
  - Si $4c\text{-}\Theta(n) \leq 0$, ó $c \leq \Theta(n)/4$
    - (valor largo de n)
  - Entonces, $T(n) = \Omega(2^n)$
  - $T(n)$ sigue siendo exponencial!

$$T(n) \geq \Theta(n) + 2\sum_{k=1}^{n-1} c2^k$$

$$= \Theta(n) + 2c\sum_{k=0}^{n-2} 2^{k+1}$$

$$= \Theta(n) + 4c\sum_{k=0}^{n-2} 2^k$$

$$= \Theta(n) + 4c(2^{n-1} - 1)$$

$$= \Theta(n) + 2c2^n - 4c$$

$$\geq c2^n$$

Memoization

- Variación de programación dinámica
  - Estrategia top-down, con el algoritmo recursivo
    - Utiliza una tabla con soluciones de subproblemas

Memoization para Multiplicación de Matrices en Cadena

- Tiempo:
  - $O(n^3)$
  - Mejor que $\Omega(2^n)$
- Memoria:
  - $O(n^2)$

```
MEMOIZED-MATRIX-CHAIN(p)
1  n ← length[p] − 1
2  for i ← 1 to n
3       do for j ← i to n
4            do m[i, j] ← ∞
5  return LOOKUP-CHAIN(p, 1, n)

LOOKUP-CHAIN(p, i, j)
1  if m[i, j] < ∞
2     then return m[i, j]
3  if i = j
4     then m[i, j] ← 0
5     else for k ← i to j − 1
6            do q ← LOOKUP-CHAIN(p, i, k)
                    + LOOKUP-CHAIN(p, k + 1, j) + p_{i−1} p_k p_j
7               if q < m[i, j]
8                  then m[i, j] ← q
9  return m[i, j]
```

Subsecuencia Común más Larga
Longest Common Subsequence (LCS)

- Una subsecuencia de otra secuencia es la misma secuencia quitándole cero o más elementos.

- Dada la secuencia $X = \langle x_1, x_2, \ldots, x_m \rangle$, otra secuencia $Z = \langle z_1, z_2, \ldots, z_k \rangle$ es una subsecuencia de X si existe una secuencia creciente $\langle i_1, i_2, \ldots, i_k \rangle$ (no necesariamente contiguos) de índices de X tal que para cada $j = 1,2, \ldots, k$, tenemos que $x_{i_j} = z_j$.

Ejemplos de Subsecuencias Comunes

- Z = <B, C, D, B> es subsecuencia de X = <A, B, C, B, D, A, B>, con la secuencia de índices <2, 3, 5, 7>

- Dadas las secuencias X y Y, Z es una secuencia común de X e Y si Z es una subsecuencia de ambas.
  - X = <A, B, C, B, D, A, B>, Y = <B, D, C, A, B, A>, la secuencia <B, C, A> es una subsecuencia común de X e Y.
  - La secuencia <B, C, B, A> es una LCS de X e Y, igual que <B, D, A, B>

Problema LCS

- Dadas dos secuencias X e Y, encontramos la subsecuencia común máxima de X e Y.

Problema LCS

- Solución por fuerza bruta
  - 2^m subsecuencias de X a buscar en Y
    - Cada secuencia es un conjunto de índices {1, 2, …, m}
  - Tiempo exponencial
  - No práctico para secuencias largas

LCS con Programación Dinámica

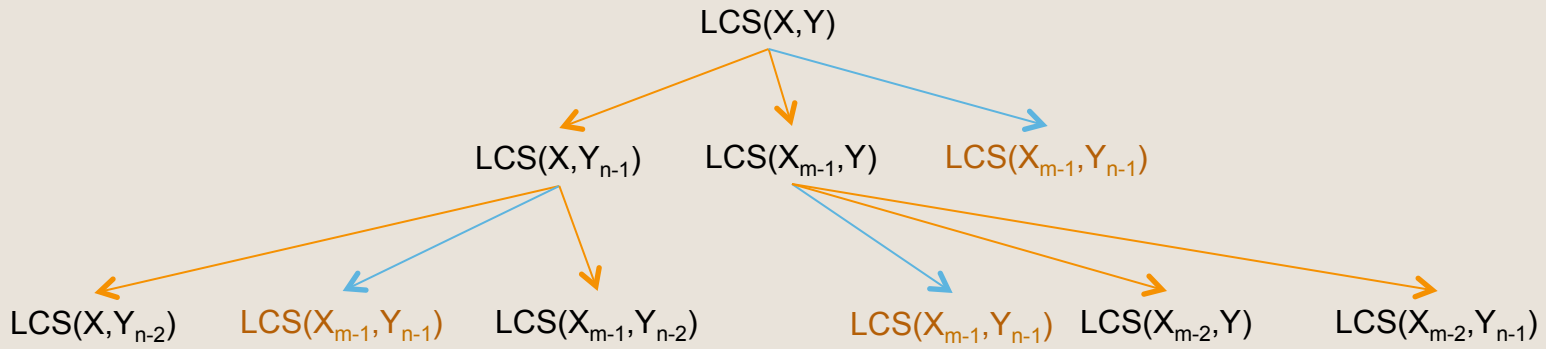
- Subestructura óptima
  - Definir
    - Dado $X = \langle x_1, \ldots, x_m \rangle$, el iésimo prefijo de X, $i = 0, \ldots, m$, es $X_i = \langle x_1, \ldots, x_i \rangle$. X_0 está vacío.
  - Teorema 16.1
    - Sean $X = \langle x_1, \ldots, x_m \rangle$ y $Y = \langle y_1, \ldots, y_n \rangle$ secuencias y $Z = \langle z_1, \ldots, z_k \rangle$ sea cualquier LCS de X y Y.

    1. Si $x_m = y_n$, entonces $z_k = x_m = y_n$ y Z_{k-1} es una LCS de X_{m-1} y Y_{n-1}
    2. Si $x_m \neq y_n$, entonces $z_k \neq x_m$ implica que Z es una LCS de X_{m-1} y Y.
    3. Si $x_m \neq y_n$, entonces $z_k \neq y_m$ implica que Z es una LCS de X y Y_{n-1}

    - Por tanto, el problema de LCS tiene subestructura óptima

LCS con Programación Dinámica

- Traslape de subproblemas
  - Sea $c[i,j]$ la longitud de una LCS en las secuencias X_i y Y_j



- La subestructura óptima del problema LCS nos lleva a la sig. fórmula recursiva

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0\,, \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j\,, \\ \max(c[i, j-1], c[i-1, j]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j\,. \end{cases}$$

LCS con Programación Dinámica

- Algoritmo recursivo exponencial para calcular longitud de una LCS de dos secuencias
  - Pero sólo hay m x n subproblemas distintos
- Solución
  - Utilizar programación dinámica
    - Procedimiento bottom up
    - En $c[i,j]$ guardamos la longitud del arreglo
    - En $b[i,j]$ guardamos el caso relacionando x_i, y_j, y z_k

Pseudocódigo LCS

LCS-LENGTH tiene un orden de $O(mn)$

```
LCS-LENGTH(X, Y)
 1  m ← length[X]
 2  n ← length[Y]
 3  for i ← 1 to m
 4       do c[i, 0] ← 0
 5  for j ← 0 to n
 6       do c[0, j] ← 0
 7  for i ← 1 to m
 8       do for j ← 1 to n
 9            do if x_i = y_j
10                 then c[i, j] ← c[i − 1, j − 1] + 1
11                      b[i, j] ← "↖"
12                 else if c[i − 1, j] ≥ c[i, j − 1]
13                      then c[i, j] ← c[i − 1, j]
14                           b[i, j] ← "↑"
15                      else c[i, j] ← c[i, j − 1]
16                           b[i, j] ← "←"
17  return c and b
```

	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

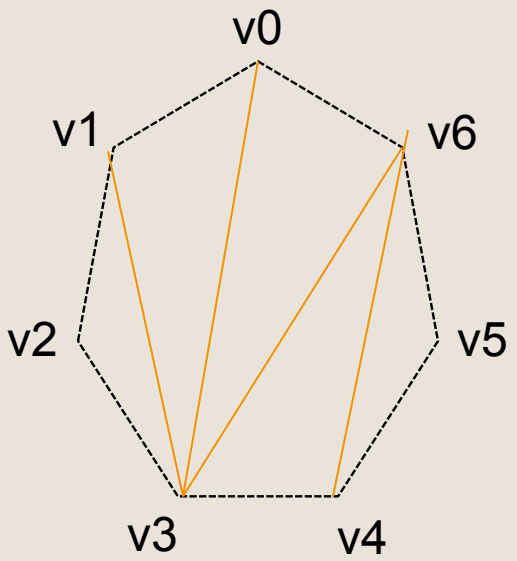
Construcción de una LCS

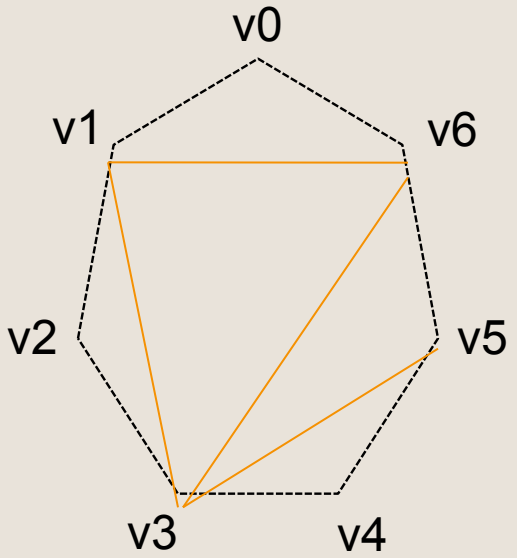
- PRINT-LCS tiene un orden de $O(m + n)$
  - Para construir la LCS
    - Iniciar en b[m,n]
    - Seguir las flechas
    - Flecha ↖ indica $x_i = y_j$, es un elemento de la LCS
  - Llamado: LCS(b,X,length[X], length[Y])

```
PRINT-LCS(b, X, i, j)
1  if i = 0 or j = 0
2     then return
3  if b[i, j] = "↖"
4     then PRINT-LCS(b, X, i − 1, j − 1)
5          print x_i
6  elseif b[i, j] = "↑"
7     then PRINT-LCS(b, X, i − 1, j)
8  else PRINT-LCS(b, X, i, j − 1)
```

Triangulación Óptima de Polígono
Optimal Polygon Triangulation

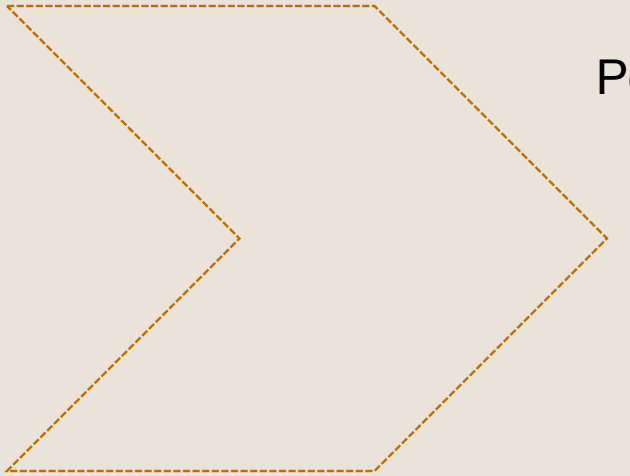
- Un polígono se define como $P = <v_0, v_1, \ldots, v_{n-1}>$





Triangulación Óptima de Polígono
Optimal Polygon Triangulation

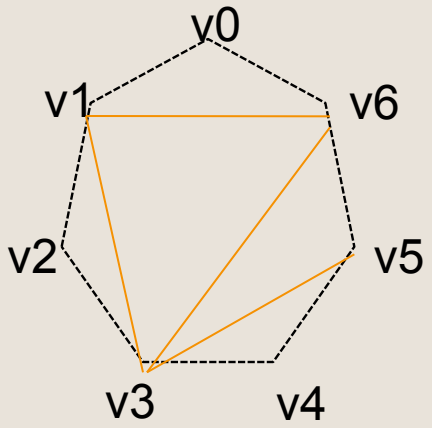
- Un polígono es convexo si un segmento de línea entre dos puntos, o en su interior, caen ya sea en sus bordes o en su interior.



Polígono no-convexo

Triangulación Óptima de Polígono
Optimal Polygon Triangulation

- Si v_i y v_j no son adyacentes, entonces el segmento $v_i v_j$ es una cuerda
- Una triangulación es un conjunto de cuerdas T que divide P en triángulos disjuntos
  - Las cuerdas no se intersectan
  - T es maximal (cada cuerda $\notin T$ intersecta una cuerda $\in T$)



Triangulación Óptima de Polígono
Optimal Polygon Triangulation

- Problema
  - Dados
    - $P = \langle v_0, v_1, \ldots, v_{n-1} \rangle$
    - Una función de pesos w sobre triángulos formada por P y T
  - Encontrar T que minimice la suma de pesos
  - Example: $w(\Delta v_i v_j v_k) = |v_i v_j| + |v_j v_k| + |v_k v_i|$ (dist. Euclidiana)
  - Este problema se puede reducir al problema de multiplicación de matrices en cadena

Triangulación Óptima de Polígono
Optimal Polygon Triangulation

- Subestructura óptima
  - T tiene $\Delta v_0 v_k v_n$
  - $w(T) = w(\Delta v_0 v_k v_n) + m[0,k] + m[k+1, n]$
  - Las dos soluciones a los subproblemas deben ser óptimas o $w(T)$ no lo sería.
  - El algoritmo requiere $\Theta(n^3)$ en tiempo y $\Theta(n^2)$ en memoria